Enabling Fast Failure Recovery in Shared Hadoop Clusters: Towards Failure-Aware Scheduling

Orcun Yildiz, Shadi Ibrahim, Gabriel Antoniu

Enabling Fast Failure Recovery in Shared Hadoop Clusters: Towards Failure-Aware Scheduling

Orcun Yildiz, Shadi Ibrahim*, Gabriel Antoniu

INRIA Rennes - Bretagne Atlantique Research Center, France

Abstract

Hadoop emerged as the *de facto* state-of-the-art system for MapReduce-based data analytics. The reliability of Hadoop systems depends in part on how well they handle failures. Currently, Hadoop handles machine failures by re-executing all the tasks of the failed machines (i.e., executing recovery tasks). Unfortunately, this elegant solution is entirely entrusted to the core of Hadoop and hidden from Hadoop schedulers. The unawareness of failures therefore may prevent Hadoop schedulers from operating correctly towards meeting their objectives (e.g., fairness, job priority) and can significantly impact the performance of MapReduce applications. This paper presents Chronos, a failure-aware scheduling strategy that enables an early yet smart action for fast failure recovery while still operating within a specific scheduler objective. Upon failure detection, rather than waiting an uncertain amount of time to get resources for recovery tasks, Chronos leverages a lightweight preemption technique to carefully allocate these resources. In addition, Chronos considers data locality when scheduling recovery tasks to further improve the performance. We demonstrate the utility of Chronos by combining it with Fifo and Fair schedulers. The experimental results show that Chronos recovers to a correct scheduling behavior within a couple of seconds only and reduces the job completion times by up to *55%* compared to state-of-the-art schedulers.

Keywords: Data Management; Scheduling; Failure; MapReduce; Hadoop, Preemption

1. Introduction

Due to its simplicity, fault tolerance, and scalability, MapReduce [1] is by far the most powerful programming model for data intensive applications. The popular open source implementation of MapReduce, Hadoop [2], was developed primarily by Yahoo!, where it processes hundreds of terabytes of data on at least *10,000 cores*, and is now used by other companies, including Facebook, Amazon, Last.fm, and the New York Times [3].

Failures are part of everyday life, especially in today's data-centers, which comprise thousands of commodity hardware and software devices. For instance, Dean [4] reported that in the first year of the usage of a cluster at Google there were around a thousand individual machine failures and thousands of hard drive failures. Consequently, MapReduce was designed with hardware failures in mind. In particular, Hadoop handles machine failures (i.e., fail-stop failure) by re-executing all the tasks of the failed machines (i.e., executing recovery tasks), by leveraging data replication.

Hadoop has not only been used for running single batch jobs, it has also recently been optimized to simultaneously support the execution of multiple diverse jobs (both batch and interactive jobs) belonging to multiple concurrent users. Several built-in schedulers (i.e., Fifo, Fair and Capacity schedulers) have been introduced in Hadoop to operate shared Hadoop clusters towards a certain objective (i.e., prioritizing jobs according to their submission times in Fifo scheduler; favoring fairness among jobs in Fair and Capacity schedulers) while ensuring a high performance of the system, mainly by accommodating these schedulers with locality-oriented strategies [5, 6, 7]. These schedulers adopt a resource management model based on *slots* to represent the capacity of a cluster: each worker in a Hadoop cluster is configured to use a fixed number of map slots and reduce slots in which it can run tasks.

*Corresponding author

Email address: shadi.ibrahim@inria.fr (Shadi Ibrahim)

While failure handling and recovery has long been an important goal in Hadoop clusters, previous efforts to handle failures have entirely been entrusted to the core of Hadoop and hidden from Hadoop schedulers. The unawareness of failures may therefore prevent Hadoop schedulers from operating correctly towards meeting their objectives (e.g., fairness, job priority) and can significantly impact the performance of MapReduce applications. When failure is detected, in order to launch recovery tasks, empty slots are necessary. If the cluster is running with the full capacity, then Hadoop has to wait until "free" slots appear. However, this waiting time (i.e., time from when failure is detected until all the recovery tasks start) can be long, depending on the cluster status (i.e., the number of free slots and the completion time of current running tasks to free more slots, if needed). As a result, a violation of scheduling objectives is likely to occur (e.g., high priority jobs may have waiting tasks while lower priority jobs are running) and the performance may significantly degrade. Moreover, when launching recovery tasks, locality is totally ignored. This in turn can further increase the job completion time due to the extra cost of transferring a task's input data through network, a well-known source of overhead in today's data-centers.

Adding failure-awareness to Hadoop schedulers is not straightforward; it requires the developer to carefully deal with challenging yet appealing issues including an appropriate selection of slots to be freed, an effective preemption mechanism with low overhead and enforcing data-aware execution of recovery tasks. *To the best of our knowledge, no scheduler explicitly coping with failures has been proposed.* To achieve these goals, this paper makes the following contributions:

- We propose Chronos[1], a failure-aware scheduling strategy that enables an early yet smart action for fast failure recovery while operating within a specific scheduler objective. Chronos takes early action rather than waiting an uncertain amount of time to get a free slot (thanks to our preemption technique). Chronos embraces a smart selection algorithm that returns a list of tasks that need to be preempted in order to free the necessary slots to launch recovery tasks immediately. This selection considers three criteria: the progress scores of running tasks, the scheduling objectives, and the recovery tasks input data locations.

- In order to make room for recovery tasks rather than waiting an uncertain amount of time, a natural solution is to kill running tasks in order to create free slots. Although killing tasks can free the slots easily, it wastes the work performed by the killed tasks. Therefore, we present the design and implementation of a novel work-conserving preemption technique that allows pausing and resuming both map and reduce tasks without resource wasting and with little overhead.

- We demonstrate the utility of Chronos by combining it with two state-of-the-art Hadoop schedulers: Fifo and Fair schedulers. The experimental results show that Chronos achieves almost optimal data locality for the recovery tasks and reduces the job completion times by up to *55%* over state-of-the-art schedulers. Moreover, Chronos recovers to a correct scheduling behavior after failure detection within only a couple of seconds.

It is important to note that Chronos is not limited to Fifo or Fair scheduling and can be easily combined with other Hadoop schedulers. Moreover, our preemption technique can be used as an alternative solution to task killing or waiting and therefore can leverage the scheduling decision in Hadoop, in general.

Relationship to previous work. This paper extends our previous contribution introduced in a previous paper [8] by providing more detailed descriptions and more thorough experiments. In particular, we have substantially extended two sections: While Section 2 gives an overview of MapReduce, Hadoop, scheduling in Hadoop and its fault-tolerance mechanism, Section 8 discusses related works on scheduling, failure recovery, task preemption and data-aware task scheduling in MapReduce. Moreover, we added more details to illustrate the different preemption techniques in Hadoop: Wait, Kill, and Preemption. We add more experiments to study the effectiveness of Chronos under multiple failures and also with different numbers of jobs and nodes. We also demonstrated the effectiveness of our preemption technique by comparing Chronos against Chronos-Kill (i.e., Chronos uses kill primitive as a preemption technique instead of the work-conserving preemption technique). Moreover, we evaluated Chronos on its sensitivity to the failure injection time. Lastly, we investigated the potential benefit of launching local recovery tasks by implementing and evaluating Chronos* (i.e., Chronos applies an aggressive slot allocation).

[1] From Greek philosophy, the god of time.

The paper is organized as follows: Section 2 presents the background of our study. We present the Chronos scheduling strategy in Section 3 and our work-conserving preemption technique in Section 4. We then present our experimental methodology in Section 5 which is followed by the experimental results in Section 6. Section 7 discusses the performance impact of Chronos. Finally, Section 8 reviews the related work and Section 9 concludes the paper.

2. Background

We provide a brief background on MapReduce, Hadoop, scheduling in Hadoop and its fault-tolerance mechanism.

2.1. MapReduce

MapReduce is a software framework for solving many large-scale computing problems [1, 9]. The MapReduce abstraction is inspired by the map and reduce functions, which are commonly used in functional languages. The MapReduce system allows users to easily express their computation as map and reduce functions. The map function, written by the user, processes a key/value pair to generate a set of intermediate key/value pairs and the reduce function, also written by the user, merges all intermediate values associated with the same intermediate key.

2.2. Hadoop

Hadoop, an open source implementation of MapReduce, is used to process massive amounts of data on clusters. Users submit jobs as consisting of two functions: map and reduce. These jobs are further divided into tasks which is the unit of computation in Hadoop. Input and output of these jobs are stored in a distributed file system. Each input block is assigned to one map task and composed of key-value pairs. In the map phase, map tasks read the input blocks and generate the intermediate results by applying the user defined map function. These intermediate results are stored on the compute node where the map task is executed. In the reduce phase, each reduce task fetches these intermediate results for the key-set assigned to it and produces the final output by aggregating the values which have the same key.

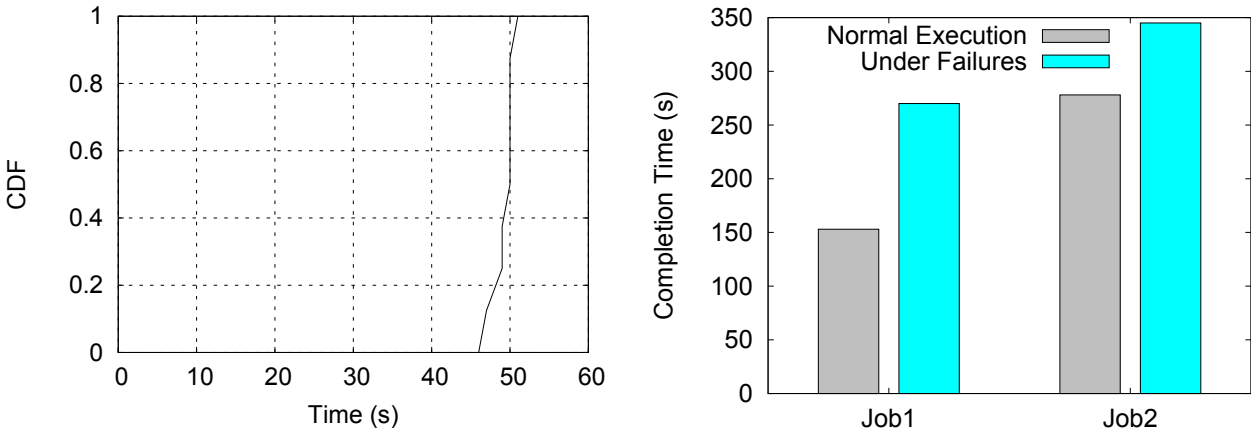
2.3. Job scheduling in Hadoop

In Hadoop, job execution is performed with a master-slave configuration. JobTracker, Hadoop master node, schedules the tasks to the slave nodes and monitors the progress of the job execution. TaskTrackers, slave nodes, run the user defined map and reduce functions upon the task assignment by the JobTracker. Each TaskTracker has a certain number of map and reduce slots which determines the maximum number of map and reduce tasks that it can run. Communication between master and slave nodes is done through heartbeat messages. At every heartbeat, TaskTrackers send their status to the JobTracker. Then, JobTracker will assign map/reduce tasks depending on the capacity of the TaskTracker and also by considering the locality of the map tasks (i.e., among the TaskTrackers with empty slots, the one with the data on it will be chosen for the map task).

The first version of Hadoop comes with a fixed Fifo scheduler. In Fifo scheduling, the JobTracker simply pulls jobs from a single job queue. Although the scheduler's name suggests the prioritization of old jobs, Fifo scheduler also takes into account jobs' priority. Hadoop is also augmented with the Fair scheduler for multi-tenant clusters. The Fair scheduler assigns resources to jobs in a way such that, on average over time, each job gets an equal share of the cluster's resources. Short jobs are able to access the cluster resources, and will finish intermixed with the execution of long jobs. Fair scheduler is primarily developed by Facebook, and aims at providing better responsiveness for short jobs, which are the majority at Facebook production clusters [6].

2.4. Fault-tolerance in Hadoop

When the master is unable to receive heartbeat messages from a node for a certain amount of time (i.e., failure detection timeout), it will declare this node as failed [10, 11]. Then, currently running tasks on this node will be reported as failed. Moreover, completed map tasks also will be reported as failed since these outputs were stored on that failed node, not in the distributed file system as reducer outputs. For a better recovery from the failures, Hadoop will try to execute the recovery tasks on any healthy node as soon as possible. To do so, Hadoop gives the highest priority to recovery tasks. Thus, recovery tasks are re-inserted in the head of the job's queue. Yet, the launching of these tasks depends on the cluster status (i.e., the number of current free slots and the completion time of current running tasks to free more slots, if needed). When there is a free slot in the cluster, Hadoop first will launch the cleanup task for the recovery task. Cleanup tasks try to ensure that failure will not affect the correct execution of the MapReduce job. When the cleanup task is completed, recovery task can run on a node with the empty slot.



(a) CDF of waiting times of recovery tasks

(b) Job completion times

Figure 1: **Recovery task waiting times and job completion times of Fifo scheduler under failure.** The experimental setup is the same as in Section 6.1.1.

3. Chronos

3.1. Design Principles

We designed Chronos with the following goals in mind:

- **Enabling an early action upon failure**: Hadoop handles failures by scheduling recovery tasks to any available slots. However, available slots might not be freed up as quickly as expected. When the Hadoop cluster is fully utilized (i.e., available slots are fully occupied) and several jobs are sharing the cluster's resources, recovery tasks will need to wait for the resources to be freed. This waiting time varies according to (1) the running applications (i.e., by analyzing the traces collected from three different research clusters [12], we observe that the execution time of map and reduce tasks varies from 2 to 84631 seconds and from 9 to 81714 seconds, respectively); and (2) the progress of the application. Consequently, this introduces uncertainty in the waiting time for launching recovery tasks. As shown in Figure 1(a), the waiting time varies from *46* to *51 seconds* which leads to increase in the completion time of jobs (see Figure 1(b)). Furthermore, during this time, scheduling objectives are violated. Chronos thus takes immediate action to make room for recovery tasks upon failure detection rather than waiting an uncertain amount of time.

- **Minimal overhead**: For the early action, a natural solution is to kill the running tasks in order to free slots for recovery tasks. Although the killing technique can free the slots easily, it results in a huge waste of resources: it discards all of the work performed by the killed tasks. Therefore, Chronos leverages a work-conserving task preemption technique (Section 4) that allows it to stop and resume tasks with almost zero overhead. On the other hand, Chronos relies on Hadoop's heartbeat messages to collect the useful information (i.e., real-time progress reports) that is fed later to our smart slot allocation strategy. The overhead of the progress reports is very little since Chronos leverages the information already provided by heartbeat messages.

- **Data-aware task execution**: Although data locality is a major focus during failure-free periods, locality is totally ignored by Hadoop schedulers when launching recovery tasks (e.g., our experimental result reveals that Hadoop achieves only *12.5%* data locality for recovery tasks, more details are given in Section 6.1.2). Chronos thus strongly considers local execution of recovery tasks.

- **Performance improvement**: Failures can severely impact Hadoop's performance: *Dinu et al.* reported that the performance of one single MapReduce application degrades by up to 3.6X for one single machine failure [10]. For multiple jobs, as shown in Figure 1(b), the job completion times increase by *30%* to *70%*. Through eliminating the waiting time to launch recovery tasks and efficiently exposing data-locality, Chronos not only

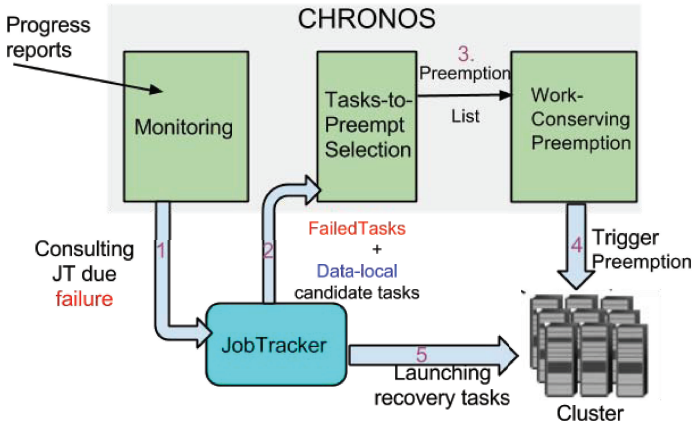


Figure 2: **Chronos overview**

corrects the scheduling behavior in Hadoop after failure but also improves the performance (i.e., reduces job completion times).

- **Resource utilization**: Chronos aims at having better resource utilization by two key design choices. First, it reduces the need for remote transfer of input data by launching local recovery tasks. Second, Chronos uses a work-conserving preemption technique that prevents Chronos from wasting the work done by preempted tasks.

- **Schedulers independent**: Chronos targets to make Hadoop schedulers failure-aware and is not limited to Fifo or Fair schedulers. Taken as a general failure-aware scheduling strategy, Chronos can be easily integrated with other scheduling policies (e.g., priority scheduling with respect to the duration of jobs). Moreover, our preemption technique can be used as an alternative solution to task killing or waiting and therefore can leverage the scheduling decision in Hadoop in general.

Hereafter, we will explain how Chronos achieves the above goals. We will discuss how Chronos allocates the necessary slots to launch recovery tasks, thanks to the tasks-to-preempt selection algorithm.

3.2. Smart slots allocation

Figure 2 illustrates the architecture of Chronos and its main functionalities. Chronos tracks the progress of all running tasks using the cost-free real-time progress reports extracted from the heartbeats. When failure is detected, Chronos consults the JobTracker to retrieve the list of failed tasks and the nodes that host their input data. Chronos then extracts the list of candidate tasks (running tasks) that belong to nodes where the input data of failed tasks reside. This list is then fed to the tasks-to-preempt selection algorithm (*Algorithm 1*) which first sorts the tasks according to the job priority. After sorting the tasks for preemption, the next step is to decide whether a recovery task can preempt any of these tasks in the sorted list. To respect scheduling objectives, we first compare the priorities of the recovery task and candidate tasks for preemption. If this condition holds, the recovery task can preempt the candidate task. For example, recovery tasks with higher priority (e.g., tasks belonging to earlier submitted jobs for Fifo or belonging to a job with a lower number of running tasks than its fair share for Fair scheduler) would preempt the selected tasks with less priority. Consequently, Chronos enforces priority levels even under failures. The list is then returned to Chronos, which in turn triggers the preemption technique in order to free slots to launch recovery tasks. If the scheduler behavior is corrected, Chronos stops preempting new tasks.

Remarks. **(1)** The main objective of Chronos is to correct the behavior of Hadoop schedulers after failure. Therefore, once this is achieved, Chronos operates similarly to Hadoop's original mechanism to avoid preempting tasks from the same job. However, given that the work-conserving preemption technique has a very low overhead and to demonstrate the potential benefit of launching local recovery tasks, in Section 6.8 we have evaluated Chronos with an aggressive

Algorithm 1: Tasks-to-preempt Selection Algorithm

```
Data: L_tasks, a list of running tasks of increasing priority;
      T_r, a list of recovery tasks t_r of decreasing priority
Result: T_p, a list of selected tasks to preempt
for Task t_r:T_r do
    for Task t:L_tasks do
        if t belongs to job with less priority compared to t_r;
        AND !T_p.contains(t) then
            T_p.add(t);
        end
    end
end
```

task preemption (noted as Chronos*). Chronos* frees slots to re-execute all recovery tasks, even within the same job. **(2)** Chronos waits until new heartbeats are received to feed the tasks-to-preempt selection algorithm with more up-to-date information. Relying on previous heartbeats information may result in preempting almost-complete tasks. This waiting time, introduced by Chronos, ranges from milliseconds to several seconds, according to the heartbeat interval[2] (*0.3 seconds* in our experiments) and the current network latency. **(3)** In general, the number of preempted tasks depends on the number of the failed tasks, the number of free slots, and the scheduler objective which determines the priority of failed tasks. At most, this number will be equal to the number of failed tasks. To have as many preempted tasks as failed tasks, all the failed tasks have to be able to preempt tasks that have lower priority than them. If there are already empty slots in the cluster, recovery tasks can be launched on these empty slots and the remaining tasks will preempt the tasks if there are tasks with lower priority than them. In this work, we consider fail-stop machine failures. Hence, the number of the failed tasks that Hadoop can tolerate is limited by the replication scheme. The maximum number of the preempted tasks is bounded by the replication scheme as well since this will be equal to the number of the failed tasks (in the worst case). With respect to the replication scheme, there can be a maximum of $R-1$ machine failures where R represents the number of replicas. The maximum number of the failed tasks will be equal to the multiplication of $R-1$ and the number of tasks belonging to currently running jobs on each failed machine. These tasks include completed and currently running map tasks on the failed machines and currently running reduce tasks on the failed machines. To note that, completed map tasks on the failed machines are considered as failed tasks since their output is not stored on the HDFS unlike the reduce tasks.

4. Work-Conserving Preemption

Preemption has been widely studied and applied for many different use cases in the area of computing. Similarly, Hadoop can also benefit from the preemption technique in several cases (e.g., achieving fairness, better resource utilization or better energy efficiency). However, only kill technique is available in Hadoop which can be used by developers as a preemption technique. Although kill technique is simple and can take a fast action (i.e. deallocating the resources), it wastes the resources by destroying the on-going work of the killed tasks. This amount of wasted work will even increase with the long running tasks. Moreover, this amount can be more significant if the killed task is a reduce task: Later, when the copy of the killed task is launched, it has to fetch all the intermediate mapper outputs which will result in additional usage of network resources that have already been scarce.

Apart from killing, also waiting approach can be taken. It is simply waiting for running tasks that should be preempted to finish, meaning that not taking any action for achieving the several goals as we described above. Waiting

[2] Hadoop adjusts the heartbeat interval according to the cluster size. The heartbeat interval is calculated so that the JobTracker receives a *min_heartbeat_interval* of 100 heartbeat messages every second (by default). Moreover, the *min_heartbeat_interval* limits the heartbeat interval to not be smaller than 300 milliseconds, corresponding to the cluster size of 30 nodes. The heart beat interval in Hadoop is computed as:
$heartbeat\_interval = \max((1000 \times cluster\_size \div number\_heartbeat\_per\_second), min\_heartbeat\_interval)$

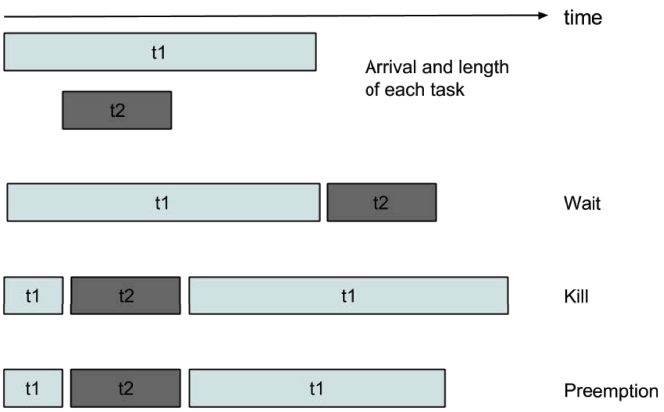


Figure 3: **Overview of the three preemption techniques**

can be efficient when a cluster runs many short jobs. However, it can introduce a considerable amount of delay for the preempting jobs in case of long-running tasks that have to be preempted.

Besides wait and kill approaches, we can also apply work-conserving preemption technique to achieve our goals. Interestingly, this technique is neither supported by Hadoop nor has been investigated by the scientific community in detail. For clarity, we illustrate these three techniques (i.e., wait, kill and preemption) in Figure 3. It presents a scenario where a shorter task needs a slot from the longer one as it arrives in the middle of the execution of the longer task. With wait approach, we can see that the execution time of the short task prolonged as much as the remaining time of the longer task to complete. With kill approach, we observe the longest execution time where on-going work of the first job is wasted and needs to be re-executed after the completion of the short task. In the last scenario with the work-conserving preemption approach, we can see that all the work that has been done by the longer task has been conserved at the moment of the preemption. After short task finishes its execution, long task continues its execution where it left off. This scenario promises the lowest average waiting time for both jobs which motivated us to leverage it within Chronos. In this paper, Chronos leverages it for better handling of failures by pausing the running tasks to make room for the recovery tasks, and resuming them from their paused state when there is an available slot.

For the preemption mechanism, a naive checkpointing approach [13] can be taken as a strawman solution. Although this approach is simple to implement, it will introduce a large overhead since checkpointing requires flushing all the data associated with the preempted task. Moreover, obtaining this checkpoint upon resume can consume a significant amount of network and I/O resources. In this section, we introduce our lightweight preemption technique for map and reduce task preemption.

4.1. Map Task Preemption

During the map phase, TaskTracker executes the map tasks that are assigned to it by the JobTracker. Each map task processes a chunk of data (input block) by looping through every key-value pair and applying the user defined map function. When all the key-value pairs have been processed, JobTracker will be notified as the map task is completed and the intermediate map output will be stored locally to serve the reducers.

For the map task preemption, we introduce an *earlyEnd* action for map tasks. The map task listens for the preemption signal from Chronos in order to stop at any time. Upon receiving the preemption request, this action will stop the looping procedure and split the current map task into two subtasks. The former subtask covers all the key-value pairs that have been processed before the preemption request comes. This subtask will be reported back to the JobTracker as completed as in the normal map task execution. On the other hand, the second subtask contains the key-value pairs that have not been processed yet. This subtask will be added to the map pool for later execution when there is an available slot, as for new map tasks. This map task preemption mechanism is illustrated in Figure 4. Full parallelism of map tasks by having independent key-value pairs gives us the opportunity to have fast and lightweight map preemption and also ensures the correctness of our preemption mechanism.

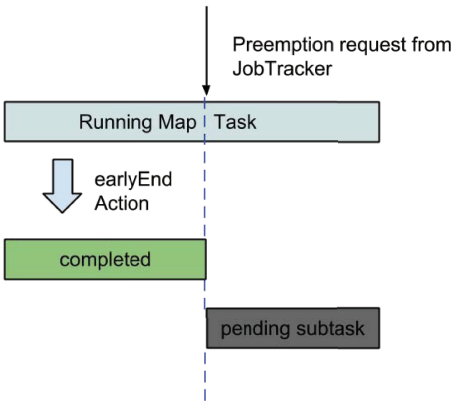


Figure 4: **Map Task Preemption**

4.2. Reduce Task Preemption

In Hadoop, reduce task execution consists of three phases: shuffle, sort and reduce. During the shuffle phase, reducers obtain the intermediate map outputs for the key-set assigned to them. The sort phase performs a sort operation on all the fetched data (i.e., intermediate map outputs in the form of key-value pairs). Later, the reduce phase produces the final output of the MapReduce job by applying the user defined reduce function on these key-value pairs.

For the reduce preemption, the splitting approach (as in the map preemption) would not be feasible due to the different characteristics of map and reduce tasks. Full parallelism of map execution and having map inputs on the distributed file system enables us to apply a splitting approach for map preemption. However, the three different phases of the reducer are not fully independent of each other. Therefore, we opt for a pause and resume approach for the reduce preemption. In brief, we store the necessary data on the local storage for preserving the state of the reduce task with pause and we restore back this information upon resume.

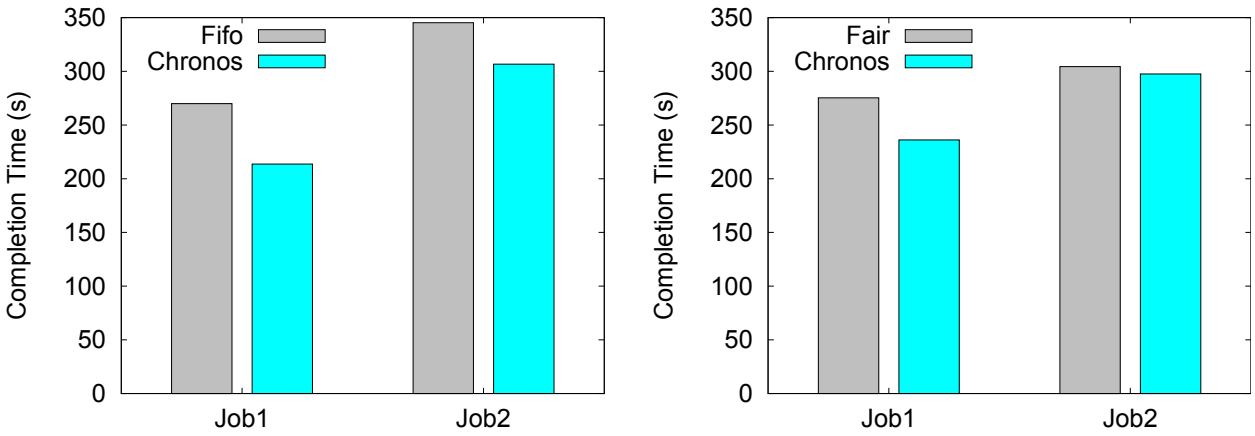
Our reduce preemption mechanism can preempt a reduce task at any time during the shuffle phase and at the boundary of other phases. The reason behind this choice is that usually the shuffle phase covers a big part of the reduce task execution, while the sort and reduce phases are much shorter. In particular, the sort phase is usually very short due to the fact that Hadoop launches a separate thread to merge the data as soon as it becomes available.

During the shuffle phase, reducers obtain the intermediate map outputs for the key-set assigned to them. Then, these intermediate results are stored either on the memory or local disk depending on the memory capacity of the node and also the size of the fetched intermediate results [14]. Upon receiving a preemption request, the pause action takes place and first stops the threads that fetch the intermediate results by allowing them to finish the last unit of work (i.e., one segment of the intermediate map output). Then, it stores all the in-memory data (i.e., number of copied segments, number of sorted segments) to local disk. This information is kept in files that are stored in each task attempt's specific folder, which can be accessed later by the resumed reduce task.

Preemption at the boundary of the phases follows the same procedure as above. The data necessary to preserve the state of the reduce task is stored on the local disk and then the reduce task will release the slot by preempting itself. The task notifies the JobTracker with a status of *suspended*. Suspended tasks will be added to the reduce pool for later execution when there is an available slot.

5. Experimental Methodology

We implemented Chronos in Hadoop-1.2.1. We evaluated Chronos performance on the Grid'5000 [15, 16] testbed, more specifically we employed nodes belonging to the parapluie cluster on Rennes site of Grid'5000. These nodes are outfitted with *12-core* AMD *1.7 GHz* CPUs and *48 GB* of RAM. Intra-cluster communication is done through a *1 Gbps* Ethernet network.



(a) Chronos *vs* Fifo scheduler

(b) Chronos *vs* Fair scheduler

Figure 5: **Performance comparison for Map-Heavy jobs**

Hadoop deployment. We configured and deployed a Hadoop cluster using *9 nodes*. The Hadoop instance consists of the NameNode and the JobTracker, both deployed on a dedicated machine, leaving *8 nodes* to serve as both DataNodes and TaskTrackers. The TaskTrackers were configured with *8 slots* for running map tasks and *4 slots* for executing reduce tasks. At the level of HDFS, we used a chunk size of *256 MB* due to the large memory size in our testbed. We set a replication factor of *2* for the input and output data. As suggested in several studies in the literature [10], we set the failure detection timeout to a smaller value (i.e., *25 seconds*) compared to the default timeout of *600 seconds*, since the default timeout is too big compared to the likely completion time of our workloads in failure-free periods.

Workloads. We evaluated Chronos using two representative MapReduce applications (i.e., wordcount and sort benchmarks) with different input data sizes from the PUMA datasets [17]. Wordcount is a Map-heavy workload with a light reduce phase, which accounts for about *70%* of the jobs in Facebook clusters [18]. On the other hand, sort produces a large amount of intermediate data which leads to a heavy reduce phase, therefore representing Reduce-heavy workloads, which accounts for about *30%* of the jobs in Facebook clusters [18].

Failure injection. To mimic the failures, we simply killed the TaskTracker and DataNode processes of a random node. We can only inject one machine failure since Hadoop cannot tolerate more failures due to the replication factor of *2* for HDFS.

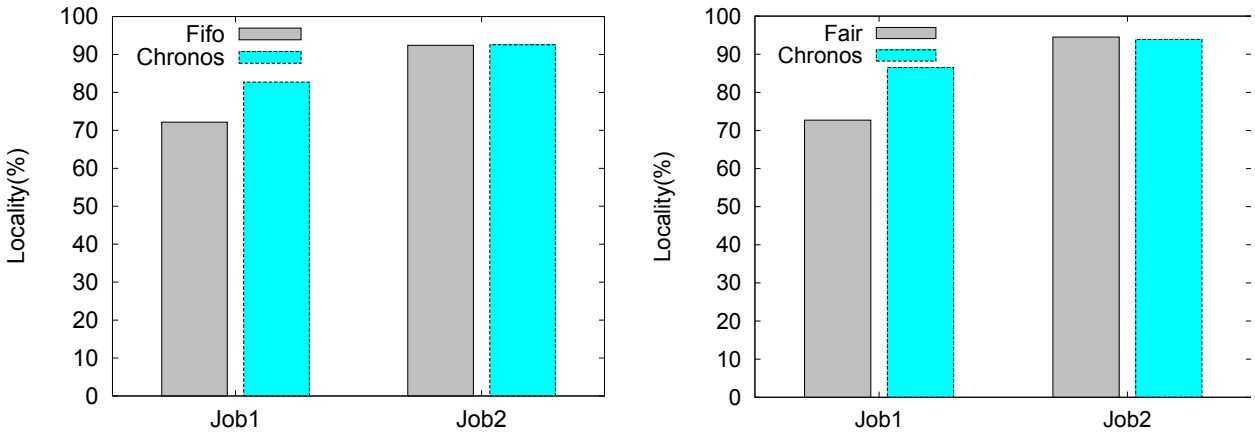
Chronos implementation. We implemented Chronos with two state-of-the-art Hadoop schedulers: Fifo (i.e., priority scheduler with respect to job submission time) and Fair schedulers. We compare Chronos to these baselines. The Fifo scheduler is the default scheduler in Hadoop and is widely used by many companies due to its simplicity, especially when the performance of the jobs is the main goal. On the other hand, the Fair scheduler is designed to provide fair allocation of resources between different users of a Hadoop cluster. Due to the increasing numbers of shared Hadoop clusters, the Fair scheduler also has been exploited recently by many companies [6, 19]. Although we implemented Chronos in Hadoop 1.2.1, the same logic can also be applied to the next generation of Hadoop, YARN [20]. Major difference of YARN from the Hadoop 1.2.x versions is that it separates the JobTracker into ResourceManager and ApplicationManager. The main motivation behind this new design is to provide better fault tolerance and scalability. However, our initial experiences with YARN indicate that severe impact of failures still exists since its schedulers are also unaware of failures and YARN adopts the same fault tolerance mechanism as Hadoop 1.2.x.

6. Experimental Results

6.1. The Effectiveness of Chronos

6.1.1. Reducing job completion time

Fifo Scheduler. We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The input data sizes result in fairly long execution times of both jobs, which allowed us to thoroughly monitor how both



(a) Data locality in Chronos and Fifo scheduler

(b) Data locality in Chronos and Fair scheduler

Figure 6: **Data locality for Map-Heavy jobs under Chronos, Fifo and Fair Schedulers**

Hadoop and Chronos handle machine failures. More importantly, this mimics a very common scenario when small and long jobs concurrently share a Hadoop cluster [6, 21]. Also, we tried to ensure that the cluster capacity (64 map slots in our experiments) is completely filled. After submitting the jobs, we have injected the failure before the reduce phase starts in order to have only map tasks as failed tasks. In contrast to Fifo, Chronos reduces the completion time of the first job and the second one by *20%* and *10%*, respectively. Most of the failed tasks belong to the first job and therefore Chronos achieves better performance for the first job compared to the second one. The main reason for the performance improvement is the fact that Fifo waits until there is a free slot before launching the recovery tasks, while Chronos launches recovery tasks shortly after failure detection. The waiting time for recovery tasks is *51 seconds* (15% of the total job execution) in the Fifo scheduler and only *1.5 seconds* in Chronos (Chronos waited *1.5 seconds* until new heartbeats arrived). Moreover, during this waiting time, recovery tasks from the first submitted job (high priority job) are waiting while tasks belonging to the second job (low priority) are running tasks. This obviously violates the Fifo scheduler rule. Therefore, the significant reduction in the waiting time not only improves the performance but also ensures that Fifo operates correctly towards its objective.

Fair scheduler. We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The failure is injected before the reduce phase starts. Figure 5(b) demonstrates that Chronos improves the job completion time by *2%* to *14%*, compared to Fair scheduler. This behavior stems from eliminating the waiting time for recovery tasks besides launching them locally. We observe that failure results in a serious fairness problem between jobs with Hadoop's Fair scheduler: this fairness problem (violation) lasts for almost *48 seconds* (16% of the total execution time) in the Fair scheduler, while Chronos restores fairness within about *2 seconds* by preempting the tasks from the jobs which exceed their fair share.

Discussion. One may think that preempting tasks from low priority jobs to launch recovery tasks of high priority jobs will obviously result in a performance degradation for low priority jobs in Chronos compared to both the Fifo and Fair schedulers. However, the performance improvements (i.e., the reductions in completion times) of high priority jobs in Chronos (due to the waiting time reduction and data locality improvement) result in an earlier release of slots and therefore tasks belonging to low priority jobs are launched earlier and fewer tasks are competing with them for resources.

6.1.2. Improving data locality

Besides the preemptive action, Chronos also tries to launch recovery tasks locally. Figure 6 shows the data locality of each job from previous experiments with Fifo (Figure 6(a)) and Fair (Figure 6(b)) schedulers. While the second job has a similar data locality, we can clearly observe that Chronos significantly improves the data locality for the first job for both scenarios (i.e., *15%* and *22%* data locality improvement compared to Fifo and Fair schedulers, respectively).

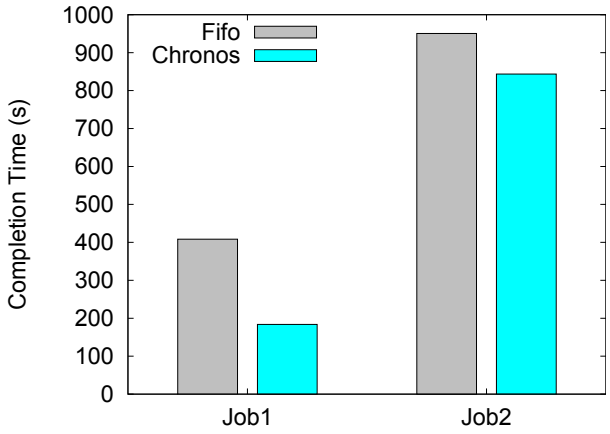


Figure 7: **Job Completion times for Reduce-Heavy jobs under Chronos and Fifo scheduler**

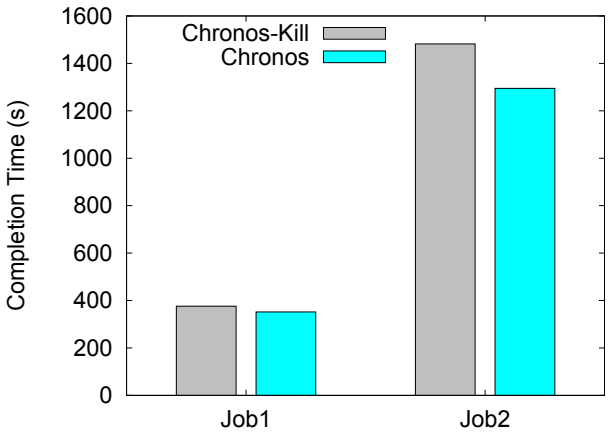
This improvement is due to the almost optimal locality for recovery tasks with Chronos (all the recovery tasks which are launched through Chronos are executed locally). Only *12.5%* of the recovery tasks were executed locally in Hadoop. The improved locality brings better resource utilization by eliminating the need for remote transfer of input blocks for recovery tasks and further improves the performance.
Summary. The aforementioned results demonstrate the effectiveness of Chronos in reducing the violation time of the scheduler (i.e., priority based on job submission time and fairness) to a couple of seconds. More importantly, Chronos reduces the completion time of the first job (the job was affected by the machine failure) due to the reduction in the waiting time and optimal locality for recovery tasks. This in turn allows the second job to utilize all available resources of the cluster and therefore improves the performance.

6.2. Impact of Reduce-Heavy Workloads

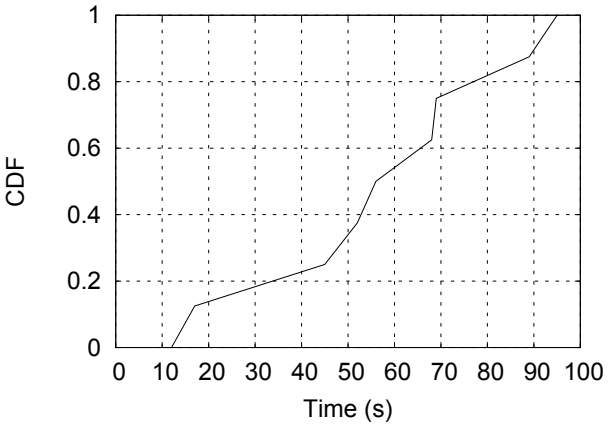
We also evaluated Chronos with Reduce-Heavy workloads. We ran two sort applications with input data sizes of *17 GB* and *56 GB*, respectively. Both jobs have *32 reduce* tasks. We injected the failure during the reduce phase in order to have failed reduce tasks from the first job. Figure 7 details the job completion time with Chronos and Fifo. Chronos achieves a *55%* performance improvement for the first job and *11%* for the second one. The improvement in the Reduce-Heavy benchmark is higher compared to the Map-Heavy benchmark because reduce tasks take a longer time until they are completed and therefore the recovery (reduce) tasks have to wait almost *325 seconds* in Fifo. Chronos successfully launches recovery tasks within *2 seconds*.
Summary. Chronos achieves a higher improvement when the failure injection is in the reduce phase which clearly states that the main improvement is due to the reduction in the waiting time. Here it is important to mention that other running reduce tasks will be also affected by the waiting time as they need to re-fetch the lost map outputs (produced by the completed map tasks on the failed machine).

6.3. The effectiveness of the preemption technique

To assess the impact of the preemption technique on Chronos performance, we have also implemented Chronos with a kill primitive as a preemption technique (Chronos-Kill). Instead of pausing the candidate tasks to free the slot, we kill the candidate tasks with Chronos-Kill. We repeated the same experiments as above, and Figure 8(a) shows the results. Although both implementations have similar completion times for first job, Chronos-Kill degrades the completion time of the second job by *12.5%*. Note that with Fifo scheduler, the first job has a higher priority compared to the second job due to its earlier submission time. Chronos-Kill thus kills the tasks from the second job to allocate slots for recovery tasks. This results in a waste of resources (we have observed that more than *50%* of the killed tasks are killed after *60 seconds* of execution, as shown in Figure 8(b)). On the other hand, our preemption mechanism has a work-conserving behavior in which the preempted tasks from the second job continue their execution without wasting the work that has already been done.



(a) Job Completion times



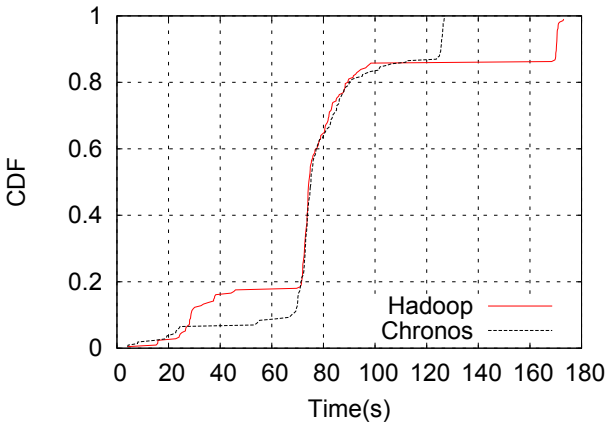
(b) CDF of running times of killed tasks, including map and reduce tasks

Figure 8: **Job Completion times for Reduce-Heavy jobs under Chronos and Chronos-Kill**

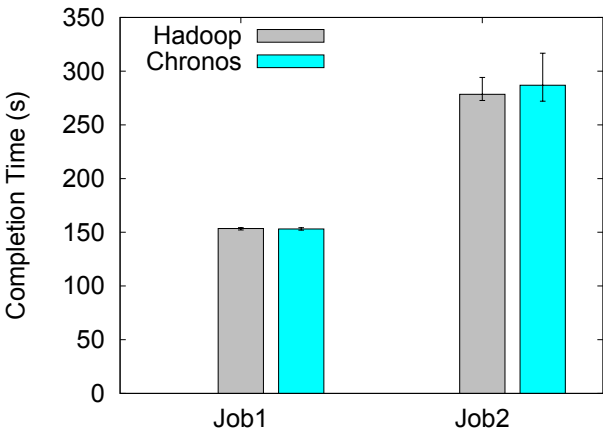
6.4. Overhead of Chronos

The overhead of Chronos may be caused by two factors: first, due to the collection of the useful information (i.e., real-time progress reports) that is fed later to our smart slot allocation strategy, and second, due to the overhead of the preemption technique. With respect to the slot allocation strategy, the overhead of Chronos is very little because Chronos leverages the information already provided by heartbeat messages. We have studied the overhead of the preemption technique by repeating the same experiment as in Section 6.1.1 (under failures). Figure 9(a) shows the completion times of each successful task with Chronos and Hadoop, we can see that they both have a similar trend. Thus, we conclude that the preemption technique does not add any noticeable overhead to cluster performance in general.

What is more, we studied the overhead of Chronos during the normal operation of the Hadoop cluster. We ran the same experiment as in Section 6.1.1 five times without any failures and Figure 9(b) shows that Chronos incurs negligible performance overhead during the normal operation.



(a) CDFs of completion times of successful tasks under Chronos and Fifo scheduler



(b) Overhead of Chronos during normal operation

Figure 9: **Overhead of Chronos**

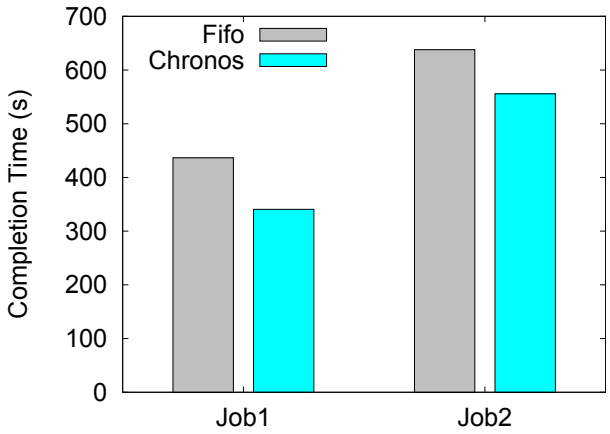


Figure 10: **Performance comparison of MapReduce jobs under double failure**

6.5. Chronos and Hadoop under multiple failures

We also evaluated Chronos under multiple failures. We repeated the same Hadoop deployment in previous experiments by only changing the replication factor of HDFS to 3 in order to tolerate two failures.

We ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. After submitting the jobs, we have injected two failures before the reduce phase starts in order to have only map tasks as failed tasks. Figure 10 illustrates that Chronos again reduces the completion time of the jobs by *22%* and *13%* thanks to its early yet smart slot allocation strategy for recovery tasks.

6.6. Chronos at scale

We evaluated Chronos with different numbers of nodes and jobs to further show its effectiveness. On 17 nodes, we ran different numbers of wordcount applications (i.e., 4, 8 and 16 jobs) with input data sizes of *17 GB* and *56 GB*. After submitting the jobs, we have injected the failure before the reduce phase starts. Table 1 shows the average job completion time and the completion time of all jobs (i.e., the time between launching the first job and the completion of the last job in the job pool) for each scenario. The improvements of the average job completion time and the completion time of all jobs with Chronos are also shown in parentheses. The results show that Chronos is able to improve MapReduce applications' performance under failures regardless of the number of jobs. Especially, we can see that Chronos brings the largest improvement in the average job completion time for the run with 4 jobs. This is expected since the number of failed tasks is almost the same regardless of the number of submitted jobs. Therefore, the improvement in the average job completion time is less significant with the increasing number of jobs. On the other hand, we observe that the run with 16 jobs has the best improvement rate (25%) with Chronos in terms of the completion time of all jobs. This is because Chronos results in earlier release of slots and therefore the tasks belonging to later jobs are launched earlier.

Table 1: Average and total completion time of MapReduce jobs on 17 nodes

Number of jobs	**Average job completion time**		**Completion time of all jobs**	
	Hadoop	*Chronos*	*Hadoop*	*Chronos*
4 jobs	306.5 s	256 s (16%)	350.1 s	323.5 s (8%)
8 jobs	350.5 s	330.9 s (6%)	410.1 s	379.3 s (7%)
16 jobs	314.4 s	297.3 s (5%)	540.1 s	406.7 s (25%)

We also ran 8 wordcount applications with input data sizes of *17 GB* and *56 GB* on different numbers of nodes: 9, 17 and 33 nodes. For 33 nodes, we used paravance cluster of the Rennes site since parapluie cluster consists of

only 19 nodes. The nodes in paravance cluster are outfitted with two 8-core Intel Xeon 2.4 GHz CPUs and 128 GB of RAM. We leverage the 10 Gbps Ethernet network for intra-cluster communication. After submitting the jobs, we have injected the failure before the reduce phase starts. Table 2 shows the average job completion and the completion time of all jobs for each different numbers of nodes. The improvement of the average job completion and completion time of all jobs with Chronos are also shown in parentheses. These results combined with the previous results demonstrate that Chronos is able to improve the performance of MapReduce applications under failures at different scales. In particular, we observe that the run with 33 nodes has the best improvement rate with Chronos in terms of average job completion time. This is because the large capacity of the cluster results in a high number of candidate tasks for preemption in Chronos. On the other hand, this behavior is exactly opposite for the completion time of all jobs where we see that the run with 9 nodes has the best improvement rate.

Table 2: Average and total completion time of MapReduce jobs on different number of nodes

Number of nodes	**Average job completion time**		**Completion time of all jobs**	
	Hadoop	*Chronos*	*Hadoop*	*Chronos*
9 nodes	730.6 s	639 s (12%)	1564.1 s	1298.2 s (17%)
17 nodes	350.5 s	330.9 s (6%)	410.1 s	379.3 s (7%)
33 nodes	205.1 s	177.6 s (14%)	245.8 s	233.1 s (5%)

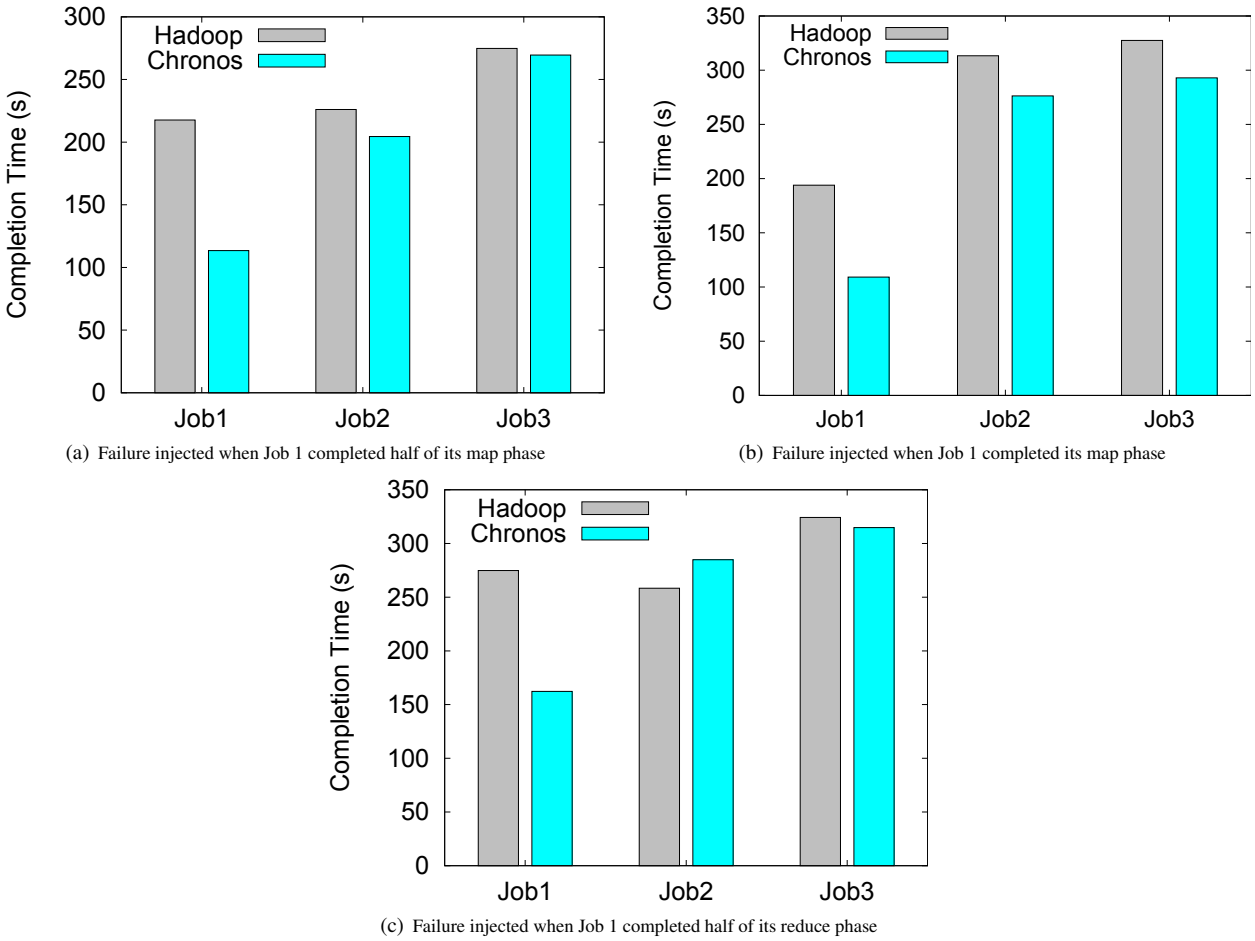
6.7. The sensitivity of Chronos to failure injection time

To assess the impact of failure injection time on Chronos performance, we performed an experiment with different failure injection times. To do so, we ran three wordcount applications with input data sizes of the first job as *17 GB* and the later jobs as *56 GB* on 17 nodes. All jobs have 64 reduce tasks and Fifo scheduling objective is used by Hadoop and Chronos. Figure 11 shows the results for three different failure injection times with respect to the first job: (1) Job 1 completed only half of its map phase, (2) Job 1 finished completely its map phase and (3) Job 1 completed half of its reduce phase when failure was injected. These results show us that Chronos reduces the job completion times even under different failure injection times with respect to the first job (i.e., the most affected job from the failure). In particular, Chronos brings the largest improvement for the first job in all three cases; this is due to the fact that the first job has most of the failed tasks compared to the other jobs when the failure was injected. Moreover, we observe that the first job has the worst performance when the failure was injected when it completed half of its reduce phase (case 3). This is because the number of the failed tasks is the highest here since failure phase covers the reduce tasks and the map tasks. Furthermore, we observe that the second job has a higher completion time with Chronos in this case, as shown in Figure 11(c). Note that the number of reduce tasks of each job is equal to the number of reduce slots in the cluster (i.e., each node has 4 reduce slots and therefore there are 64 reduce slots in total.). When Chronos preempts the reducers from the second job to allocate resources for the recovery reduce tasks of the first job, these preempted tasks have to wait for free slots. Therefore, the second job has a performance degradation with Chronos in order to favor the execution of the recovery tasks. However, this performance degradation (10%) is much lower compared to the performance improvement gained for the first job (40%).

6.8. Chronos with aggressive slot allocation

After observing that Chronos's preemption technique has a negligible overhead, we slightly changed the smart slot allocation strategy of Chronos by implementing Chronos* with aggressive slot allocation strategy. With Chronos, recovery tasks with higher priority would preempt the selected tasks with less priority. With Chronos*, we also allow recovery tasks to preempt the selected tasks with the same priority (e.g., recovery tasks belonging to the same job with selected tasks). One may think that preempting tasks from the same job to launch recovery tasks would not improve the job performance. However, thanks to the work-conserving preemption technique we can safely preempt the tasks even belonging to the same job in order to improve the locality of recovery tasks.

To assess the effectiveness of Chronos*, we ran two wordcount applications with input data sizes of *17 GB* and *56 GB*, respectively. The failure is injected before the reduce phase starts. We adjusted the network speed to *256 Mbps* in order to incorporate the locality factor on the performance. The results are shown in Figure 12. Although both



(a) Failure injected when Job 1 completed half of its map phase

(b) Failure injected when Job 1 completed its map phase

(c) Failure injected when Job 1 completed half of its reduce phase

Figure 11: **Job completion times of Hadoop and Chronos under different failure injection phases**

implementations have similar completion times for the first job, Chronos* degrades the completion time of the second job by *17%* compared to Chronos. Note that with Chronos, recovery tasks that belong to the second job would not be able to preempt any of the running tasks due to their later submission time. However, we observed 100% locality for recovery tasks with Chronos* thanks to its aggressive slot allocation strategy by also allowing the recovery tasks to preempt the tasks from the same job.

7. Discussion

The reliability of Hadoop systems depends in part on how well they tolerate failures. There are two major factors in achieving a better fault tolerance: failure detection and failure handling. Failure detection timeout plays an important role in the fault tolerance mechanism as it directly impacts the first factor, failure detection. However, this is out of scope of this work. With Chronos, we target to achieve better handling of the failures. We observe throughout our experiments that we achieve this by mainly reducing the waiting time for the recovery tasks. Although, failure detection timeout has a direct impact for the failure detection, this does not hold for the failure handling case. To note that, reduction in the waiting time is equal to the time between the failure detection and completion time of the current running tasks. Hence, failure detection timeout only indicates the start for the waiting time of the recovery tasks while the duration of the currently running tasks marks the finish time. By analyzing the traces collected from three different research clusters [12], we observe that the average execution time of map and reduce tasks is 124 *seconds*

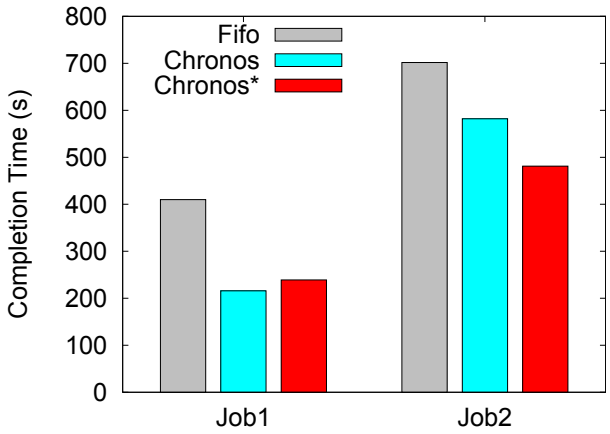


Figure 12: **Job Completion times under Fifo, Chronos and Chronos* schedulers with single failure**

and 901 *seconds*, respectively. Thus, having long running tasks, despite the failure injection and detection times, is common in Hadoop clusters. Therefore, Hadoop is vulnerable to incur serious performance degradations under fail-stop failures [10, 22, 23]. Our results demonstrate that Chronos recovers to a correct scheduling behavior within a couple of seconds only and reduces the job completion times (i.e., improves the performance of MapReduce jobs). The performance improvement is due to two main factors: the reduction in the waiting time to launch recovery tasks and the improved locality of recovery tasks. The reduction in waiting time varies according to the status and the progress speed of running tasks when detecting the failure. However, as shown in many studies, recently Hadoop cluster is shared by multiple different MapReduce applications (i.e., differ in their complexity and thus in the average execution time of their tasks [6], [21]. As a result, there is a significant potential for improving the performance of jobs using Chronos. Moreover, we observe that Chronos does not only improve the performance of the high-priority jobs but also eases the execution of the low-priority jobs. This stems from the fact that performance improvements of high priority jobs in Chronos result in an earlier release of slots and therefore tasks belonging to low priority jobs are launched earlier and fewer tasks are competing with them for resources. On the other hand, network is normally the most scarce resource in today's data-centers [24]. By launching local recovery tasks, Chronos reduces the extra cost for data transferring of these sensitive tasks. Therefore, Chronos not only eliminates the waiting time but also improves the recovery tasks' execution times by launching them locally.

8. Related Work

Scheduling in MapReduce. There exists a large body of studies on exploring new objectives (e.g., fairness, job priority) when scheduling multiple jobs in MapReduce and improving their performance. *Isard et al.* introduced Quincy [19], a fair scheduler for Dryad, which treats scheduling as an optimization problem and uses min-cost flow algorithm to achieve the solution. Quincy uses the kill mechanism to set the cluster according to the configuration in the solution. *Zaharia et al.* introduced a delay scheduler [6], a simple delay algorithm on top of the default Hadoop Fair scheduler. Delay scheduling leverages the fact that the majority of the jobs in production clusters are short, therefore when a scheduled job is not able to launch a local task, it can wait for some time until it finds the chance to be launched locally. More recently, *Venkataraman et al.* have proposed KMN [25], a MapReduce scheduler that focuses on applications with input choices and exploits these choices for performing data-aware scheduling. KMN also introduces additional map tasks to create choices for reduce tasks. This in turn results in less congested links and better performance. Although these scheduling policies can improve the MapReduce performance, none of them is failure-aware, leaving the fault tolerance mechanism to the MapReduce system itself, and thus are vulnerable to incurring uncertain performance degradations in case of failures. Moreover, Chronos can complement these policies to enforce correct operation and to further improve their performance under failures.

Failure recovery in MapReduce. Due to failure being one of the characteristics of MapReduce environments, several studies have been dedicated to explore and improve the performance of MapReduce systems under failures. *Dinu et*

al. [10] have demonstrated a large variation in Hadoop job completion time in the presence of failure. In particular, because Hadoop uses the same functionality to recover from TaskTracker failure regardless the cause/type of such failures (i.e., permanent failure in case of node failure or temporary failure in case of TCP connection failure due to network contention). *Ruiz et al.* have proposed RAFT [22], a family of fast recovery algorithms upon failures. RAFT introduces checkpointing algorithms to preserve the work upon failures. However, an experimental study is performed only with a single job scenario. *Dinu et al.* have proposed RCMP as a first-order failure resilience strategy instead of data replication [26]. RCMP performs efficient job recomputation upon failures by only recomputing the necessary tasks. However, RCMP only focuses on I/O intensive pipelined jobs, which makes their contribution valid for a small subset of MapReduce workloads. Our work is different in the targeting environment, as we focus on shared Hadoop clusters with multiple concurrent jobs.

Exploring and exploiting task preemption in MapReduce. There have been a few studies on introducing work-conserving preemption techniques to MapReduce environments. *Wang et al.* [27] exploited the fact that long running reduce tasks may lead to starvation of short jobs. Thus, they introduced a technique for reduce task preemption in order to favor short jobs against long jobs. Similar to our pause and resume approach, they preserve the state of a reduce task upon pause and restore this necessary task state upon resume. However, the preemption technique is limited to reduce tasks and there is no overhead study regarding the preemption technique. Moreover, they leverage the preemption only for fairness while we can leverage Chronos for different scheduling objectives. *Liu et al.* introduced PDCS [28], Preemptive Deadline Constraint Scheduler, which aims at minimizing the completion time of jobs under deadlines. To do so, PDCS employs the kill primitive of Hadoop in order to allocate slots for the tasks that belong to near deadline jobs. Although PDCS can reduce the deadline misses in the job pool, it incurs a significant amount of wasted work due to leveraging the kill primitive. *Pastorelli et al.* [29] have proposed a preemption technique with a pause and resume mechanism to enforce the job priority levels for the job execution. For the pause and resume mechanism, they leverage the already available POSIX signals such as SIGSTP and SIGCONT. These signaling mechanisms can suspend the running tasks since tasks in Hadoop are Unix processes running in JVMs. Although these mechanisms can simplify the preemption, it brings the shortcoming that a suspended task can only be launched on the same machine on which it was paused before. Therefore, their preemption technique is not work-conserving at all times since suspended tasks will start their execution from scratch if the job scheduler assigns them on a different node than they were paused before. In addition, they only use a simple scheduler which fetches the task eviction policies from a static configuration file for evaluating their system. Actually, this work supports our work in demonstrating the importance of preemption to leverage job scheduling in Hadoop. *Ananthanarayanan et al.* [30] introduced Amoeba to support a lightweight checkpointing mechanism for reduce tasks with the aim of achieving better elasticity for resource allocation. Also, this preemption technique ignores the map task preemption as [27]. However, we observe that having long running map tasks is also common and it is necessary to consider them for better resource allocation or performance depending on the preemption objective. In contrast, Chronos introduces both map and reduce task preemption and leverages it to make Hadoop schedulers failure-aware.

Data-aware task scheduling. Several research efforts have been made with the aim of having better locality for achieving higher performance [31, 5, 32]. *Ibrahim et al.* have proposed Maestro [31], a replica-aware map scheduler for Hadoop that tries to increase locality in map phase and also yields better balanced intermediate data distribution for shuffle phase. *Ananthanarayanan et al.* introduced Scarlett [5], a system that employs a popularity-based data replication approach to prevent machines that store popular content from becoming bottlenecks, and to maximize locality. Chronos also aims at maximizing locality but it mainly focuses on recovery tasks that belong to different jobs.

9. Conclusion

Hadoop has emerged as a prominent tool for Big Data processing in large-scale clouds. Failures are inevitable in large-scale systems, especially in shared environments. Consequently, Hadoop was designed with hardware failures in mind. In particular, Hadoop handles machine failures by re-executing all the tasks of the failed machine. Unfortunately, the efforts to handle failures are entirely entrusted to the core of Hadoop and hidden from Hadoop schedulers. This may prevent Hadoop schedulers from meeting their objectives (e.g., fairness, job priority, performance) and can significantly impact the performance of the applications.

We address this issue through the design and implementation of a new scheduling strategy called Chronos. Chronos is conductive to improving the performance of MapReduce applications by enabling an early action upon failure detection. Chronos tries to launch recovery tasks immediately by preempting tasks belonging to low priority jobs, thus avoiding the uncertain time until slots are freed. Moreover, Chronos strongly considers the local execution of recovery tasks. The experimental results indicate that Chronos results in almost optimal locality execution of recovery tasks and improves the overall performance of MapReduce jobs by up to *55%*. Chronos achieves that while introducing very little overhead.

Thanks to these encouraging results, we plan to further investigate the potential benefits of the work-conserving preemption technique. In particular, Hadoop schedulers are still relying on wait or kill primitive to ensure the QoS requirements of several users; thus an interesting direction to explore is how to ensure QoS requirements without wasting the cluster resources. Moreover, we are currently exploring a new scheduling policy that relies on our preemption technique for improving the energy efficiency of Hadoop clusters.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `http://www.grid5000.fr/`).

References

[1] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Communications of the ACM 51 (2008) 107–113.
[2] The Apache Hadoop Project, `http://www.hadoop.org`, Accessed on Sep 2015.
[3] Powered By Hadoop, `http://wiki.apache.org/hadoop/PoweredBy`, Accessed on Sep 2015.
[4] J. Dean, Large-scale distributed systems at google: Current systems and future directions, in: Keynote speach at The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, Big Sky, MT, USA, 2009.
[5] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris, Scarlett: coping with skewed content popularity in mapreduce clusters, in: Proceedings of the sixth ACM European Conference on Computer Systems (EuroSys'11), 2011, pp. 287–300.
[6] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, in: Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys'10), 2010, pp. 265–278. doi:`http://doi.acm.org/10.1145/1755913.1755940`.
[7] C.-H. Hsu, K. D. Slagter, Y.-C. Chung, Locality and loading aware virtual machine mapping techniques for optimizing communications in mapreduce applications, Future Generation Computer Systems 53 (2015) 43 – 54.
[8] O. Yildiz, S. Ibrahim, T. A. Phuong, G. Antoniu, Chronos: Failure-Aware Scheduling in Shared Hadoop Clusters, 2015. URL: `https://hal.inria.fr/hal-01203001`, in the 2015 IEEE International Conference on Big Data (IEEE BigData 2015).
[9] H. Jin, S. Ibrahim, L. Qi, H. Cao, S. Wu, X. Shi, The mapreduce programming model and implementations, Cloud Computing: Principles and Paradigms (2011) 373–390.
[10] F. Dinu, T. E. Ng, Understanding the effects and implications of compute node related failures in hadoop, in: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12), 2012, pp. 187–198.
[11] D. Huang, X. Shi, S. Ibrahim, L. Lu, H. Liu, S. Wu, H. Jin, Mr-scope: a real-time tracing tool for mapreduce, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, Illinois, 2010, pp. 849–855. doi:`http://doi.acm.org/10.1145/1851476.1851598`.
[12] Hadoop Workload Analysis, `http://www.pdl.cmu.edu/HLA/index.shtml`, Accessed on Sep 2015.
[13] K. Salem, H. Garcia-Molina, Checkpointing memory-resident databases, in: Proceedings of the Fifth International Conference on Data Engineering (ICDE'89), IEEE, 1989, pp. 452–462.
[14] S. Ibrahim, T.-D. Phan, A. Carpen-Amarie, H.-E. Chihoub, D. Moise, G. Antoniu, Governing energy consumption in hadoop through cpu frequency scaling: An analysis, Future Generation Computer Systems (2015) –.
[15] Grid'5000 Home page, `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`, 2015.
[16] Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, T. Iréa, Grid'5000: a large scale and highly reconfigurable experimental Grid testbed., International Journal of High Performance Computing Applications 20 (2006) 481–494.
[17] F. Ahmad, S. Lee, M. Thottethodi, T. Vijaykumar, Puma: Purdue Mapreduce benchmarks suite, ECE Technical Reports. Paper 437 (2012).
[18] Y. Chen, S. Alspaugh, R. Katz, Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads, Proceedings of the VLDB Endowment 5 (2012) 1802–1813.
[19] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09), 2009, pp. 261–276.
[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al., Apache hadoop yarn: Yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing (SoCC'13), 2013, pp. 1–16.
[21] K. Ren, Y. Kwon, M. Balazinska, B. Howe, Hadoop's adolescence: An analysis of hadoop usage in scientific workloads, Proc. VLDB Endow. 6 (2013) 853–864.

[22] J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, J. Dittrich, Rafting mapreduce: Fast recovery on the raft, in: Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE'11), 2011, pp. 589–600.
[23] S. Ibrahim, T. A. Phuong, G. Antoniu, An eye on the elephant in the wild: A performance evaluation of hadoop's schedulers under failures, in: Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC-2015), held in conjunction with PODC'15, 2015.
[24] J. Schad, J. Dittrich, J. Quiané-Ruiz, Runtime measurements in the cloud: Observing, analyzing, and reducing variance, PVLDB 3 (2010) 460–471.
[25] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, I. Stoica, The power of choice in data-aware cluster scheduling, in: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation, USENIX Association, 2014, pp. 301–316.
[26] F. Dinu, T. Ng, Rcmp: Enabling efficient recomputation based failure resilience for big data analytics, in: 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS'14), 2014, pp. 962–971. doi:10.1109/IPDPS.2014.102.
[27] Y. Wang, J. Tan, W. Yu, X. Meng, L. Zhang, Preemptive reducetask scheduling for fair and fast job completion, in: Proceedings of the 10th International Conference on Autonomic Computing, ICAC, volume 13, 2013.
[28] L. Liu, Y. Zhou, M. Liu, G. Xu, X. Chen, D. Fan, Q. Wang, Preemptive hadoop jobs scheduling under a deadline, in: Semantics, Knowledge and Grids (SKG), 2012 Eighth International Conference on, IEEE, 2012, pp. 72–79.
[29] M. Pastorelli, M. Dell'Amico, P. Michiardi, Os-assisted task preemption for hadoop, arXiv preprint arXiv:1402.2107 (2014).
[30] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, I. Stoica, True elasticity in multi-tenant data-intensive compute clusters, in: Proceedings of the Third ACM Symposium on Cloud Computing (SoCC'12), ACM, 2012, p. 24.
[31] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Maestro: Replica-aware map scheduling for mapreduce, in: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'12), IEEE, 2012, pp. 435–442.
[32] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, L. Qi, Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud, in: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CLOUDCOM'10), Indianapolis, USA, 2010, pp. 17–24. doi:http://dx.doi.org/10.1109/CloudCom.2010.25.